

Object Oriented Programming (OOP) Essay

Abdulmalik Elghedafi

Thursday 17th April 2025

Contents

1	Introduction	2
2	Design of program	2
3	Inheritance	5
3.1	Best Practices	5
4	Interfaces	7
4.1	Best Practices	7
5	Abstract classes	8
5.1	Best Practices	8
6	Polymorphism	8
6.1	Best Practices	9
7	Virtual Methods	9
7.1	Best practices	9
8	Storage Structures	10
8.1	Best practices	10
9	Generics	10
10	Best practices	11
10.1	Best data structure for the car park application	11
11	Memory allocation	11
11.1	Static memory allocation	11
11.2	Dynamic memory allocation	12

12 Main	12
12.1 carpark	12
12.2 MyCarpark	13
12.3 NumberplateReader	13
12.4 IDReader	14
12.5 IDBarcodeReader	14
12.6 Barrier	14
12.7 FullSign	14
12.8 Sensor	14
13 Conclusion	15
14 Appendix	15
14.1 Guide to run program	15
15 Unit test cases	17
16 Screenshots of program working	20
17 Links	25

1 Introduction

The purpose of this report is to explore the design methodology and development details of a Java based application built using object-oriented programming principles. Based on a provided narrative and incomplete design class diagram, the application stimulates a real-world car park system, where cars enter and exit using specialised barcode readers, such as a numberplate reader, which is implemented through inheritance by deriving from the parent to subclasses. As part of this task additional classes, attributes and methods were identified and implemented to ensure full functionality.

The report examines the core object-oriented concepts including generics, polymorphism, abstract classes, interfaces, storage structures and virtual methods. These are discussed with reference to established best practices from academic literature, including journals, books and Oracle documentation. Code snippets are provided to demonstrate how the implementation adheres to these principles. Finally, the report discusses memory allocation, comparing static and dynamic approaches, outlining their benefits and limitations. It also justifies the choice made for the car park application.

2 Design of program

A flow chart design for the entry and exit of the car park OOP application is shown in (Figure 1) and (Figure 2).

Figure 1: Design for entry of carpark application

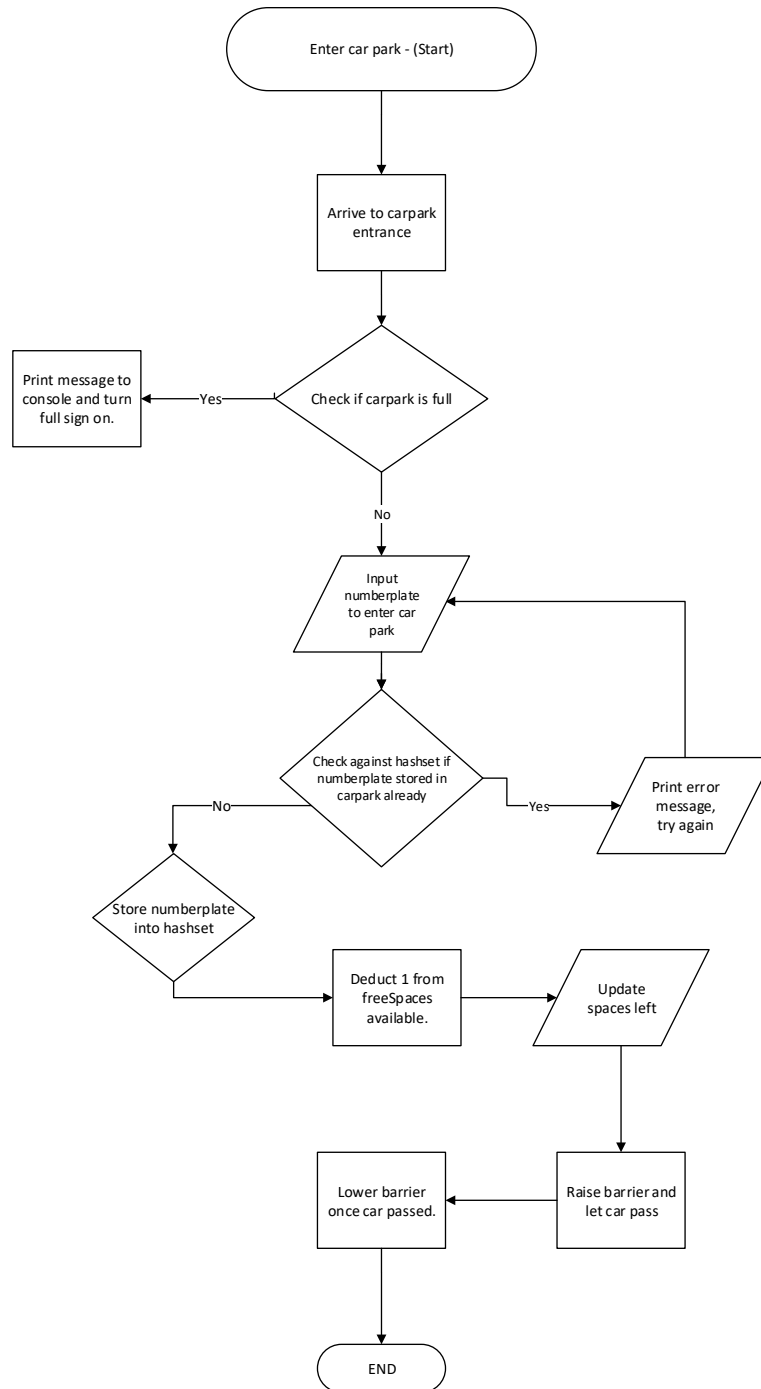
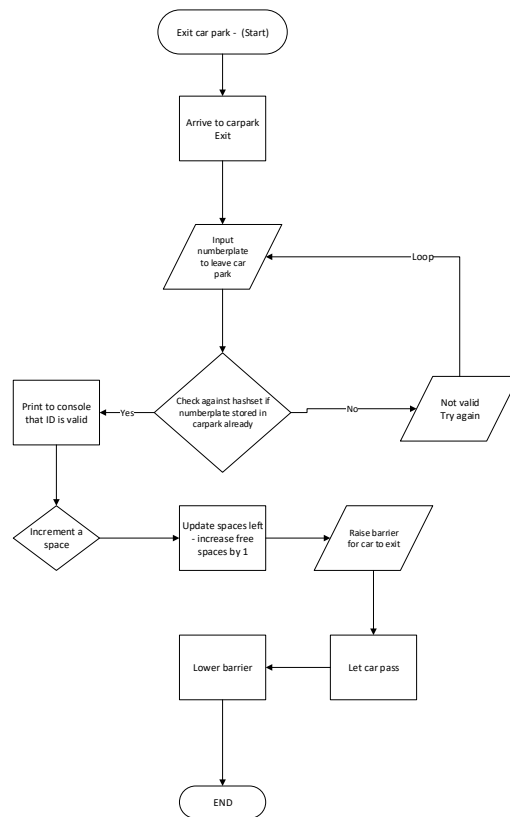


Figure 2: Design for exit of carpark application



3 Inheritance

Inheritance supports code reuse by enabling subclasses to inherit and extend the functionality from the parent class. Inheritance is justified only if a “is-a” relationship exists. It is advisable to use it only when the relationship is logically sound. For instance, a forwarded message is a subclass of message which supports the use of inheritance. In comparison a mailbox has a greeting which indicates a “has-a” relationship, thus favouring composition as the more appropriate design choice. According to Horstmann, class hierarchies are commonly represented as tree structures, where the root is the general concept and more specialised ones toward the leaf nodes. This approach supports readability and future extensibility. Albeit overly deep or poorly designed hierarchies can lead to fragile systems and cause tight coupling. It is prudent to use composition instead of inheritance when the relationship between objects does not constitute a genuine “is-a” relationship. Adopting this principle supports maintainable design and mitigates the risk of unforeseen errors when modifying base classes [1] [2].

3.1 Best Practices

```
package org.example;

// ID READER IS FOR DRIVER LICENSE SERIAL NUMBER
public abstract class IDReader<T> implements Sensor { 2 usages 2 inheritors  ⚡ Abdulmalik Elghedafi +1 *
    private String serialNumber; 2 usages

    public abstract void read(); 2 usages 2 implementations new *
    public abstract void update(); no usages 2 implementations new *

    // constructor
    public IDReader(T id) { 2 usages new *
    }

    public String getSerialNumber() { 1 usage  ⚡ Abdulmalik Elghedafi
        return serialNumber;
    }

    public void setSerialNumber(String serialNumber) { 1 usage  ⚡ Abdulmalik Elghedafi
        this.serialNumber = serialNumber;
    }
}
```

Figure 3: IDReader abstract class for inheritance.

My application uses inheritance by facilitating the definition of specialised input reader types. It uses an abstract class for IDReader shown in (Figure 3) to define shared functionality i.e. `read()` and `update()` methods, but should not be

instantiated on its own. NumberPlateReader and IDBarcodeReader (Figure 4) are both concrete subclasses which extend from the IDReader class and forces subclasses to implement the methods which makes it flexible and consistent. This establishes a strong “is-a” relationship between the parent and subclasses as they’re both specialised IDReaders.

```
package org.example;

public class NumberplateReader extends IDReader<String>{ 4 usages  ⚡ Abdulmalik Elghedafi +1 *
    private final String numberplate; 2 usages
    private final boolean carPresent; 2 usages

    // constructor
    public NumberplateReader(String numberplate) { 2 usages  ⚡ Abdulmalik Elghedafi
        super(numberplate);
        this.numberplate = numberplate;
        this.carPresent = true;
    }

    // read method
    @Override 2 usages new *
    public void read()
    {
        System.out.println("\uD83D\uDCD6 Reading number plate: " + numberplate);
    }

    @Override no usages  ⚡ Abdulmalik Elghedafi +1 *
    public boolean carPresent() {
        return true;
    }

    @Override no usages  ⚡ Abdulmalik Elghedafi +1
    public void update() {
        if (carPresent) {
            System.out.println("✅Car is Present");
        } else {
            System.out.println("❌Invalid. Car is not present!");
        }
    }
}
```

Figure 4: NumberPlateReader inheriting from the abstract IDReader class.

4 Interfaces

An effective strategy for implementing interfaces within object-oriented programming (OOP) is to adhere to the Interface Segregation Principle (ISP), thereby ensuring that clients are not dependent on methods irrelevant to their specific requirements. Tightly coupling clients through large interfaces can compromise the system and undermine the maintainability. To mitigate potential breaches of the Liskov Substitution Principle (LSP), interfaces should be carefully simplified and narrowly scoped, exposing only essential functionalities to preserve both substitutability and interface integrity. Martin emphasises that Interfaces must be cohesive, encapsulating a single, clear and well-defined abstraction. Furthermore, prioritising composition over inheritance can enhance system flexibility and support improved maintainability over time [3].

4.1 Best Practices

```
package org.example;

// Adding an interface for sensor so other subclasses can use it.
// Also, these two methods will be overridden for other classes.

public interface Sensor { 2 usages 3 implementations  ⓘ Abdulmalik Elghedafi
    boolean carPresent(); 1 usage 2 implementations  ⓘ Abdulmalik Elghedafi
    void update(); no usages 3 implementations  ⓘ Abdulmalik Elghedafi
}
```

Figure 5: The sensor class using an interface.

```
public void raise(Sensor sensor) { 2 usages ⓘ Abdulmalik Elghedafi +1
    if (sensor.carPresent()) {
        up = true;
        System.out.println("\uD83C\uDD99 - Barrier is now raised!");
    }
}
```

Figure 6: Sensor being used for the raise method in the barrier class

The Sensor class (Figure 5) defines a shared contract without specifying implementation details. The interface is implemented by the IDReader class which extends to types of other readers. For example, NumberPlateReader and ID-BarcodeReader. In the Barrier class (Figure 6) it uses the Sensor interface

irrespective of how it is implemented. Which promotes loose coupling, being a fundamental principle of effective interface design.

5 Abstract classes

In object-oriented programming, abstract classes are utilised to specify common behaviour for similar classes whilst avoiding direct instantiation. Abstract classes can include fully defined methods, instance variables and abstract methods, although the latter is required to be implemented by subclasses. Abstract classes function as a bridge between fully defined classes and interfaces, which enables shared functionality to be extracted and moved to the parent class. To facilitate code reusability and ease of maintenance, it is regarded as best practice to relocate common functionality into an abstract class. Software developers are advised to structure class hierarchies meticulously as a class can extend only one abstract class. To enhance code flexibility, it is prudent to leverage abstract classes together with interfaces as recommended by Horstmann, thus enabling classes to implement several interfaces and handle a wider variety of behaviours [1] [2].

5.1 Best Practices

The best practices for abstract classes is demonstrated in the above paragraph with inheritance.

6 Polymorphism

Polymorphism is a fundamental concept in object-oriented programming that enhances modularity by allowing objects of different classes to be accessed through a common interface while demonstrating unique behaviours. However, when combined with multi-level class inheritance and dynamic dispatch, it can introduce complexity, potentially affecting productivity and making debugging more challenging. To mitigate this, developers must manage complexity with care. A widely accepted approach to improve flexibility and reduce tight coupling is to define behaviour through interfaces or abstract classes. At runtime, dynamic binding guarantees the appropriate method for the object's actual type is invoked, thereby enabling reliable substitution of subclasses without breaking functionality. [4] [2].

6.1 Best Practices

```
// read method
@Override 2 usages new *
public void read()
{
    System.out.println("\uD83D\uDCD6 Reading number plate: " + numberplate);
}

@Override no usages ٢ Abdulmalik Elghedafi +1 *
public boolean carPresent() {
    return true;
}

@Override no usages ٢ Abdulmalik Elghedafi +1
public void update() {
    if (carPresent) {
        System.out.println("✅Car is Present");
    } else {
        System.out.println("❌Invalid. Car is not present!");
    }
}
```

Figure 7: Polymorphism used in numberplate reader

Polymorphism is used in my carpark class through (Figure 7) abstract classes defined in the IDReader base class, which are then overridden in subclasses through method overriding to implement their own logic. This promotes best practices as it adheres to the open/closed principle where the IDReader class defines the `read()` method and override the `read()` method in NumberplateReader class.

7 Virtual Methods

Virtual methods in object-oriented programming allow subclasses to override superclass behaviour, facilitating code reuse and polymorphism. Although versatile, they can lead to error-prone systems due to tightly coupled components. A best practice is to prefer composition over inheritance, as it often results in more maintainable, flexible and loosely coupled code [5].

7.1 Best practices

Virtual methods best practices has been written in the section for polymorphism.

8 Storage Structures

The collection framework, storage structures in Java is convenient as it makes data structured and data optimised. This ensures that large amount of data can be added without falling apart and guarantees the code will execute faster. It is advantageous as it hides the sophisticated internal details of how the storage structure works and provides a ready made storage structure to use.

They are made to be flexible and integrate well with any type of collection in Java. Thus, enhances code reusability. There are pro's and con's for each storage structure where some are better suited than others [6].

8.1 Best practices

```
HashSet<String> parkedCars = new HashSet<>();
```

Figure 8: HashSet implemented for the storage structure in main

A `HashSet<String>` had been used for the carpark application as shown in (Figure 8) . This is to store the numberplates inside the carpark. It meets best practice, as it checks for duplicated numberplates and ensures that if the carpark stored 5000 cars, the lookup times would be relatively the same as looking up one car. Therefore, it remains powerful and an efficient and reliable to store data as it uses $O(1)$ complexity.

9 Generics

Generics let software developers to specify which type is suitable in a collection. This permits the compiler to manage the conversion of types dynamically and notify the user if the incorrect type was applied before runtime. By using generics, this ensures code is readable and optimising type safety.

The cautions of generics can be when you utilise raw types, the type safety is bypassed, thus resulting in producing runtime errors like `ClassCastException`. It is best to avoid raw types in code and stick to generics [5].

```
public class NumberplateReader extends IDReader<String>{
```

Figure 9: Generics used in Number plate reader class

```

public abstract class IDReader<T> implements Sensor { 2 usages 2 inheritors ⓘ Abdulmalik Elghedafi +1 *
    private String serialNumber; 2 usages

    public abstract void read(); 2 usages 2 implementations new *
    public abstract void update(); no usages 2 implementations new *

    // constructor
    public IDReader(T id) { 2 usages new *
    }

```

Figure 10: Generics used in IDReader

10 Best practices

Generics is used in my application for type safety in (Figure 9) and (Figure 10) and minimise errors during runtime. The `HashSet<String>` for parked cars in main is to store the numberplates and preventing the incorrect type from being added i.e. Integer. This is a recommended approach as it catches errors early on, promoting clear and maintainable code.

10.1 Best data structure for the car park application

When designing the car park application, selecting an efficient data structure to store number plates was essential. A `HashSet` was chosen due to its constant time average performance $O(1)$ for adding, removing and checking entries within the dataset, irrespective of the size [7] [8]. It also avoids duplicate entries, to ensure that the same numberplate can not be added twice. This suits the application well for high performance and look up times.

Alternative data structures such as `ArrayList` or `LinkedList` use $O(n)$ which is linear time. This slows down the operation when searching and removing elements from the list. While a `TreeSet` automatically sorts data and restricts duplicates, it uses $O(\log n)$ which uses more memory and is significantly slower than a `HashSet`. Therefore making the `HashSet` the optimal solution for the carpark application [8].

11 Memory allocation

11.1 Static memory allocation

In Java, `Arrays` utilise static memory allocation, where the size is defined when it is created and cannot be altered during runtime. This facilitates efficient and rapid access to elements due to their consistent memory structure. That said, if additional space is required, a new array must be initialised and have its data copied over, which is ineffective. To prevent wasting memory, slots within the array should all be used. While arrays can offer improved performance and ease

of use, they lack the flexibility that dynamic structures can offer i.e. `LinkedList` which can expand on size dynamically during runtime [9].

11.2 Dynamic memory allocation

In relation to java data structures, dynamic memory allocation occurs during runtime and allows data structures to grow or shrink dynamically. This is carried out using the `new` operator, where objects are stored in the heap memory space. `ArrayList`, `HashSet`, `HashMap` and `LinkedList` are examples of data structures that make use of the dynamic allocation memory, to adjust their size based on the amount of data stored. In comparison, objects created on the heap do not get erased when a method exits, whereas local stack variables disappear. To free up space, and get rid of unused objects, the JVM (Java Virtual Machine) relies on the mark-sweep algorithm which handles cleaning up memory automatically without the programmers participation. To build good software that can grow and adapt, it is essential to use dynamic allocation, even if it comes with some inefficiencies [10].

In my car park application, I used a `HashSet` to store parked cars. The dynamic memory allocation allows the program to handle changes at runtime without establishing a fixed size array. This demonstrates the importance in building flexible and scalable software.

12 Main

The `main` class is where my application lives. In main, a `HashSet` memory based storage structure is instantiated as `parkedCars`. Then an object `carpark` is instantiated using the `MyCarPark` class, where you can set the capacity and available spaces available in the car park and whether the `fullSign` light is on/off.

The program uses a while loop for the logic of the program. It handles the input from the user's choice and prints a display menu. It has validation to ensure the user's choice is a number and not outside the range of 1 and 5. If the user enters a string, boolean or any input other than an integer, a message will be printed to notify the user. It also has a try and catch statement, to catch exception errors.

A `switch` statement is used to determine which methods should run for each class based on the option the user selected from the menu.

12.1 carpark

The `carpark` class defines several methods that are responsible for managing the availability of updating the carpark spaces and updating the status of when the carpark becomes full. All my attributes are set to private to prevent them from being modified directly from outside the class.

Attributes:

- **capacity** An integer for the max number of spaces the carpark can hold.
- **freeSpaces** An integer for the spaces currently free.
- **theCarPark** A hashset String to hold the numberplates.
- **fullSign** A fullSign field that refers to the fullSign class that switches on/off the lights.

The **incrementSpaces** method performs a conditional check whether the available spaces is lower than what the carpark's max capacity can hold. If there are no free spaces left, then a statement is printed to the console to notify the user. The **decrementSpaces** handles the reduction of carpark spaces, when a car reserves a space, it reduces one space from the available spaces and notifies the user how many spaces are left. If no available spaces are left, a statement is printed to the console and the full sign light is switched on. To achieve whether the carpark is full the **checkIfCarParkIsFull** checks if no available spaces are left and turns the fullsign method on. Alternatively the **checkIfCarParkIsNotFull** will check and return whether the carpark is not full. The **update** is an abstract method which is used in the **MyCarPark** class.

12.2 MyCarpark

The **MyCarPark** class is a subclass that inherits the properties and behavior which are defined in the **Carpark** abstract class. In this class, I use **super()** to call the constructor which ensures the fields are initialised before the subclass logic.

The **update** method overrides the abstract method from the **Carpark** class and prints a statement to the console of the available spaces left.

12.3 NumberplateReader

The **NumberplateReader** class is a subclass which inherits the properties and behaviour of the **IDReader** class. This class is responsible for managing the numberplates which is taken as input from the user in the main class.

Attributes:

- **numberplate** An string for the input of the numberplate
- **carPresent** A boolean for whether the car is present.

Both the **read** and **update** methods are abstract methods which provide concrete implementations. The **read** method reads the numberplate and is used within the main class, specifically on the **readingNumberplate** object and prints the numberplate. The **carPresent** checks whether the car is present, but was

not necessary to use in the program. The `update` method checks if the car is/not present and prints a statement in the program, but it was not necessary either.

12.4 IDReader

The `IDReader` is a base and a abstract class that uses generics and implements a `Sensor` as the interface. It contains one private field for a `serialNumber`. This provides different types of readers for the program, i.e. `NumberplateReader` and `BarcodeReader`.

12.5 IDBarcodeReader

The `IDBarcodeReader` inherits the method and behaviour from the `IDReader`. It has two private fields; `carPresent` which is a boolean and `barcode` which is a string. The `read` method reads the barcode and returns it. Then the `carPresent` checks if the car is present at reader, but I did not use it. I also have a `read` and `update` method, but did not implement these for my application, as I did not need them. The purpose of the `IDBarcodeReader` was to demonstrate my implementation.

12.6 Barrier

The `Barrier` class manages the functionality of raising and lowering the carpark barrier when a car arrives to the gate and leaves the carpark. It takes one private field `up`. In the `raise` method, it uses `sensor` as a parameter which references the `Sensor` interface in relation of the barrier sensor in the carpark. So when a car is present then the barrier is raised and the user is notified. The same goes with `lower` method, which prints that the barrier is lowered. These two methods are used in the `main` class to raise and lower the barrier.

12.7 FullSign

The `FullSign` class manages when the light should be turned on and off when the carpark reaches max capacity. It takes a boolean private field `lightOn`. The `SwitchOn` method is set to true and outputs the `fullSign` is on and the carpark is full. As well as printing that no further numberplates can be added. This method is used in the `main` class. The `switchOff` method turns the light off and makes the carpark available, but has not been used in `main`.

12.8 Sensor

The `Sensor` is an interface to allow my other class: `IDReader` to define a contract for what the class must have. It has a boolean for `carPresent` and void for `update`. The `carPresent` method is used in the `barrier` class.

13 Conclusion

This report explored the design and development of a car park application using OOP principles such as inheritance, encapsulation, polymorphism, generics, interfaces, and abstraction. These were implemented following best practices from academic journals, books, and Oracle documentation. The application models car entry and exit using inherited ID readers and notifies the user when the car park is full before entry. Dynamic memory allocation was chosen.

This application meets the project brief criteria; however, future improvements should focus on supporting ticket numbers and driver serial numbers as alternatives to number plates for entering or leaving the car park. As this was my first OOP programming project, I was able to apply previously learned theoretical concepts in a practical context. The assignment for the car park application was both engaging and rewarding. Everything began to make sense one night, and from that point on, I was able to apply OOP principles more effectively and with greater confidence in my programming.

14 Appendix

14.1 Guide to run program

To run the program, set the program to Java 21 as shown in (Figure 11). Then run `Main.java` to start the program. A menu of options will appear. You can set the size of the carpark for the free spaces of cars that can park by going to the code and changing the code from two to five as shown in (listing 1). The current program is set to two to check that the program passes all criteria set for the assingment.

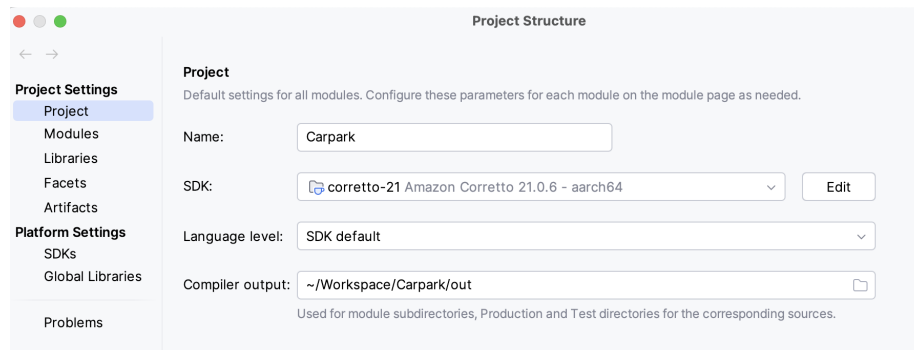


Figure 11: HashSet implemented for the storage structure in main

```
1 MyCarpark carpark = new MyCarpark(10, 5, parkedCars, new FullSign(true));
2
3
```

Listing 1: Java code snippet for HashSet size

The program menu has five options the user can select which represents a real car park application (Figure 12).

```
Welcome to my car park!
-----
1. Check if car park is full.
2. Enter a number plate to enter car park
3. Enter car park number to leave car park
4. Check what cars are parked.
5. Exit

-----
Enter your choice:
```

Figure 12: HashSet implemented for the storage structure in main

15 Unit test cases

Name of test	Method name	Class name	Description of test	Figure
raiseTest	Raise(Sensor sensor)	Barrier	Checks if the car is present and raises the barrier.	14
lowerTest()	Lower()	Barrier	Checks if the barrier is up, then lowers it down	15
incrementSpacesToIncreaseASpace	incrementSpaces	Carpark	To ensure free spaces are being incremented properly	16
decrementSpacesToDecreaseASpace	decrementSpaces	Carpark	To ensure that when a car reserves a space in the carpark, it will decrement 1 from free spaces and print a message with the updated count of spaces.	17
checkIfCarparkIsFull	checkIfCarParkIsFull	Carpark	Checks to see if free spaces is equal to zero and switches on the full sign light.	18
checkIfCarparkIsNotFullTestTrue	checkIfCarParkIsNotFull	Carpark	Checks if carpark is not full	19
checkIfCarparkIsNotFullTestFalse	checkIfCarParkIsNotFull	Carpark	Checks if carpark is full.	20
SwitchOnTest	SwitchOn	FullSign	Turns on carpark full sign on, if carpark full.	21
SwitchOffTest	switchOff	FullSign	Turns off fullsign	22
IDBarcodeReaderTest	IDBarcodeReader	IDBarcodeReader	Reads the barcode	23
CheckIfCarPresentTest	carPresent	IDBarcodeReader	Checks if car is present, car is not at barrier so returns false	24
getSerialNumberTest	getSerialNumber	IDReader	Prints the serial number	25

Figure 13: Table of unit test cases

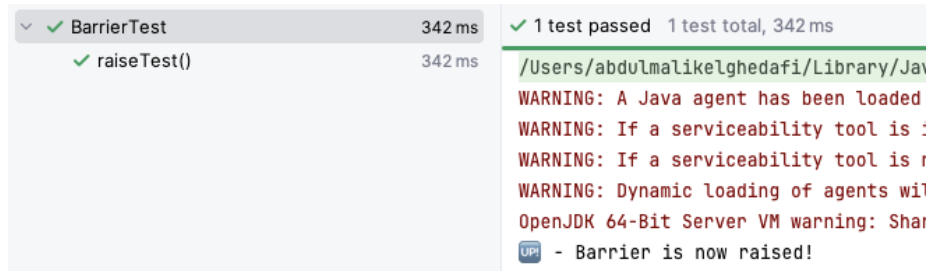


Figure 14: Checks if car is present and raises the barrier

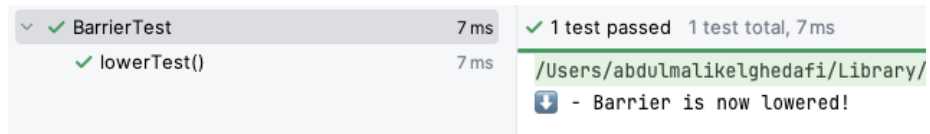


Figure 15: Checks if barrier is up and lowers it down

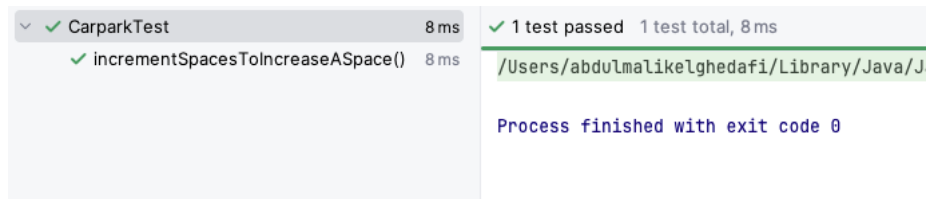


Figure 16: Increments a free space

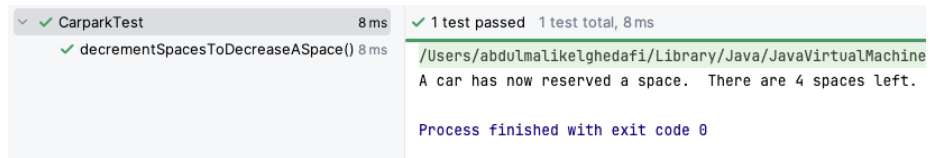


Figure 17: Decrements a free space



Figure 18: Checks if free space is equal to zero and switches on full sign on

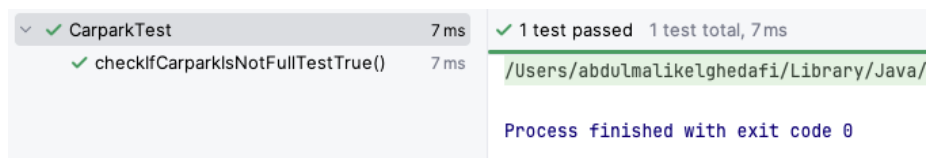


Figure 19: Checks if car park is not full

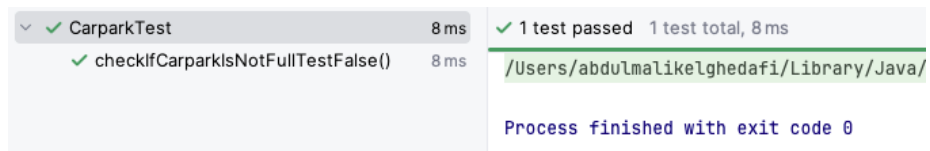


Figure 20: Checks if car park is full

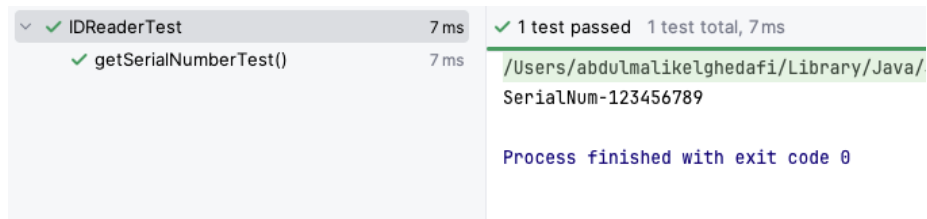


Figure 25: Prints the serial number

16 Screenshots of program working

Enter your choice: **1**

 Full sign: Off - Car park is not full.

 [UPDATE] The car park has 2 free spaces available

Welcome to my car park!

1. Check if car park is full.
2. Enter a number plate to enter car park
3. Enter car park number to leave car park
4. Check what cars are parked.
5. Exit

Enter your choice: |

Figure 26: Checking if carpark is full

Enter your choice: 2

Enter a new number plate to enter car park: test

📖 Reading number plate: test

🚧 - Barrier is now raised!

✅ Number plate: test has now been added.

⬇️ - Barrier is now lowered!

A car has now reserved a space. There are 1 spaces left.

Welcome to my car park!

1. Check if car park is full.
2. Enter a number plate to enter car park
3. Enter car park number to leave car park
4. Check what cars are parked.
5. Exit

Enter your choice:

Figure 27: Adding a numberplate to the program

Enter your choice: 4

 The Cars currently in the car park are: [test]

Welcome to my car park!

1. Check if car park is full.
2. Enter a number plate to enter car park
3. Enter car park number to leave car park
4. Check what cars are parked.
5. Exit

Enter your choice:

Figure 28: Checking what cars are parked

```
Enter your choice: 3

Enter the numberplate to leave car park: test
📖 Reading number plate: test
🚧 - Barrier is now raised!
❌ [test] Has been removed from car park
⬇️ - Barrier is now lowered!
Welcome to my car park!
-----
1. Check if car park is full.
2. Enter a number plate to enter car park
3. Enter car park number to leave car park
4. Check what cars are parked.
5. Exit

-----
Enter your choice: |
```

Figure 29: Removing a numberplate from the car park

Enter your choice: 4

🚗 The Cars currently in the car park are: []

Welcome to my car park!

1. Check if car park is full.
2. Enter a number plate to enter car park
3. Enter car park number to leave car park
4. Check what cars are parked.
5. Exit

Enter your choice:

Figure 30: Checking number plates again to show the numberplate has been removed

```

Enter your choice: 2

Enter a new number plate to enter car park: test1234566789

💡 FullSign is on: car park is full!
🔴 Sorry - can't add a new numberplate.

Welcome to my car park!
-----
1. Check if car park is full.
2. Enter a number plate to enter car park
3. Enter car park number to leave car park
4. Check what cars are parked.
5. Exit

-----
Enter your choice:

```

Figure 31: Showing that you can not add more than two plates, as set by constructor

17 Links

Implementation: <https://www.youtube.com/watch?v=Pj5jgRZHUaY>

Design: <https://www.youtube.com/watch?v=f0EiqgrubcA>

GitHub link: <https://github.com/2420753/Carpark>

References

- [1] C. S. Horstmann, *Object-Oriented Design & Patterns*, 2nd ed. Hoboken, N.J.: John Wiley & Sons, 2006.
- [2] V. Vamanmehtre and A. Bhalla, “Analysis of object oriented programming,” *IRE Journals*, vol. 3, no. 6, pp. 16–20, 2019, no DOI available. [Online]. Available: <https://www.irejournals.com/paper-details/1701782>
- [3] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Book, Ed. USA: Prentice Hall PTR, 2003.

- [4] H. Ali, “Exploring polymorphism: Flexibility and code reusability in object-oriented programming,” *Passer Journal of Basic and Applied Sciences*, vol. 6, no. Special Issue, Jan. 2024.
- [5] J. Bloch, *Effective Java*, 3rd ed. Boston: Addison-Wesley, 2018, titelzusätze auf dem Umschlag: ”Updated for Java 9. Best practices for ... the Java platform”.
- [6] H. Schildt, *Java: The Complete Reference, Ninth Edition*, 9th ed. New York: McGraw-Hill Education, 2014, ”Fully updated for Java SE 8 (JDK 8)”–Cover. - ”Oracle Press.”. - Includes index. - Description based on print version record.
- [7] Oracle. (2014) Hashset (java platform se 8). Online Documentation. [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html>
- [8] I. D. Srihith, A. D. Donald, T. A. S. Srinivas, D. Anjali, and R. Varaprasad, “The backbone of computing: An exploration of data structures,” *International Journal of Advanced Research in Science, Communication and Technology*, pp. 155–163, Apr. 2023.
- [9] N. Karumanchi, *Data Structures and Algorithms Made Easy in Java Data Structure and Algorithmic Puzzles, Second Edition*. Createspace Independent Publishing Platform, 2013.
- [10] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, *Data structures and algorithms in Java*, 6th ed. Hoboken, NJ: Wiley, 2014.